Memory Allocation
Computer Science 240
Laboratory 10

In this lab, you will be introduced to the final project for the course, implementing a dynamic memory allocator for C programs.

The repository contains:

`Makefile` – recipes for compiling
`mdriver.c` – testing driver
`memlib.h` – memory/heap interface
`mm.c` – memory allocator implementation (**your code goes here)**
`mm.h` – memory allocator interface
`traces/` – several trace files (`.rep`) used for simulated testing
the remaining files are testing support files you do not need to inspect

You will compile with **`make`** to produce an executable called **`mdriver`**.

There are a variety of ways to run the executable, described in the assignment (read carefully when you are ready to start testing your code).

Block Structure

```
                           ... lower addresses up
         HEAP BLOCK
      +--------------------+  <-- Start of block
      | header (size|tags) |
   ---+--------------------+  <-- Start of payload (when allocated)
    P |                    |      Address is multiple of ALIGNMENT
    A |                    |
    Y |        ...         |
    L |                    |
    O |                    |
    A +--------------------+
    D | footer (size)      |
   ---+--------------------+  <-- End of block, start following block.
                              ... higher addresses down
```

A **word** is 8 bytes in our machine.

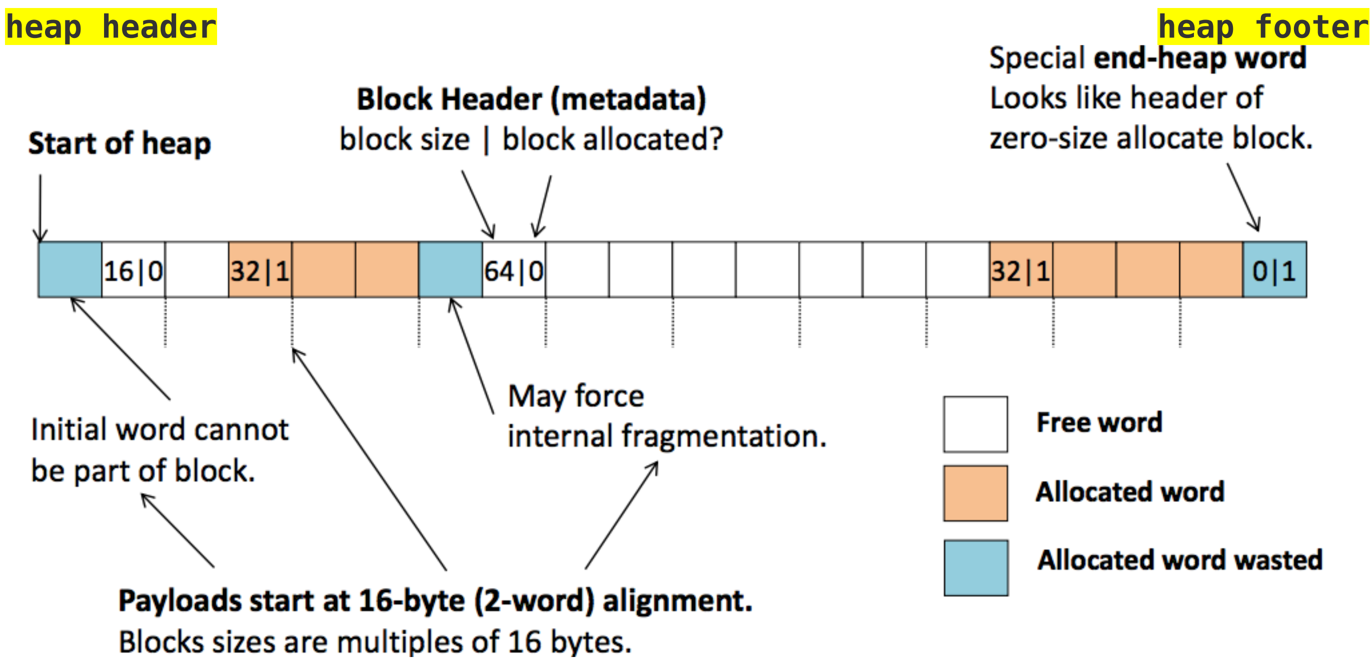
The initial word of a block is called a **block header** or a **status word,** and stores the size of the block in bytes (multiple of 16). It also stores whether the block is used, and whether the preceding block is used in the bottom two bits of the status word (called a **tag):**

The bottom two bits (**tags**) are:

```
          Bit  1 (2^1 == 2): PRED_USED_BIT tag    [For coalescing]
          Bit  0 (2^0 == 1): USED_BIT tag
```

Free blocks use the last word as a **footer,** which stores SIZE ONLY (no tags).

Heap Block Layout (Implicit Free List)



Starter code

In *mm.c*, you are given a working memory allocator implementation. When an allocation occurs, the heap is checked to see if there is a large enough free block available for the allocation. If there is not a large enough block, the head is extended by a page which contains a large new free block.

The starter code does not split when you allocate, and it does not do anything when you free (it does not free the block, and does not coalesce as part of the free, either).

Your job will be to add code to the *allocate* function to implement splitting, and to write the *mm_free* and *coalesce* functions.

The starter code initializes the heap in the *mm_init* function.

The heap is initialized to a single **page** of memory, which is 4096 bytes, and contains a single large free block, along with the **heap header** and **heap footer**.

The heap header starts at an address which is word-aligned (boundary of 8). This guarantees that the first block payload also starts at a word-aligned address.

The heap footer is a special block of size 0 with the USED_BIT set to 1. This is the only block with this configuration, so marks the end of the heap.

The first block in the heap is assumed to always have the PRED_USED_BIT set to 1 (so that you will not try to coalesce memory below the heap when you free the first block).

When an allocation occurs, the heap is checked to see if there is a large enough free block available for the allocation. If there is not a large enough block, the head is extended by a page which contains a large new free block.

The following are declarations in *mm.c* which enforce the definitions given.

Base address of the heap

```
#define HEAP_BASE ((word_t*)mem_heap_lo())
```

Bound address of the heap (first address after the heap)

```
#define HEAP_BOUND ((word_t*)PADD(mem_heap_hi(), 1))
```

Address of the heap header

```
#define HEAP_HEADER_ADDR ((word_t**)HEAP_BASE)
```

Address of the first block in the heap

```
#define ORIGIN_BLOCK_ADDR ((word_t*)PADD(HEAP_BASE, WORD_SIZE))
```

Address of the heap footer word

```
#define HEAP_FOOTER_ADDR ((word_t*)PSUB(HEAP_BOUND, WORD_SIZE))
```

Type for **word**

```
typedef unsigned long word_t;
```

Size of **word**
All pointers and size_t values are one word in size.

```
#define WORD_SIZE (sizeof(word_t))
```

ALIGNMENT
Payloads must be aligned to 2 words

```
#define ALIGNMENT ((size_t)(2*WORD_SIZE))
```

Minimum block size

```
#define MIN_BLOCK_SIZE (ALIGNMENT)
```

Functions for masking header/status word

The size and two tags can be extracted separately from the block header/status word by masking, using the following functions:

status_size(x) extracts the block size information from a status word, x, masking off the other status bits.

```
#define SIZE_MASK (~(ALIGNMENT - 1))

static word_t status_size(word_t status_word) {
     return status_word & SIZE_MASK;
}
```

status_pred(x) extracts the predecessor status bit from a status word, x, masking off the other status bits.

```
#define PRED_USED_BIT 2

static word_t status_pred(word_t status_word) {
     return status_word & PRED_USED_BIT;
}
```

status_used(x) extracts the allocation status bit from a status word, x, masking off the other status bits.

```
#define USED_BIT 1

static word_t status_used(word_t status_word) {
     return status_word & USED_BIT;
}
```

make_status(s,p,u) makes a new status word by extracting the block size information from status word s, the predecessor status bit from word p, and the allocation status bit from word u.

WARNING: to set the predecessor or used bits explicitly, pass PRED_USED_BIT or USED_BIT, not 1.

```
static word_t make_status(word_t size, word_t pred_used, word_t used) {
     return (size & SIZE_MASK) | (pred_used & PRED_USED_BIT) | (used &
     USED_BIT);
}
```

Functions for block headers

The following functions are provided for easy access/manipulation of block headers.

get_header(word_t* block) Get the header word of the block

```
static word_t block_get_header(word_t* block) {
     return LOAD(block);
}
```

block_set_header(word_t* block, word_t header) Set the header of the block

```
static void block_set_header(word_t* block, word_t header) {
     STORE(block, header);
}
```

block_succ(word_t* block) calculate the address of the block successor

```
static word_t* block_succ(word_t* block) {
     // Get this block's size from its header and add to its address.
     return PADD(block, status_size(LOAD(block)));
}
```

block_pred(word_t* block) calculate the address of the block predecessor, assuming its predecessor is free

```
static word_t* block_pred(word_t* block) {
     // Predecessor must be free.
     assert(!status_pred(LOAD(block))&& "predecessor must be free");

     // Get predecessor size from predecessor footer and subtract from
     // this block's address.
      word_t footer = LOAD(PSUB(block, WORD_SIZE));

     // Footers must hold sizes.
     assert(status_size(footer) == footer && "footer must hold
                                               size only, no status bits");
     return PSUB(block, footer);
}
```

Functions for unscaled pointer arithmetic (PADD and **PSUM)**
These functions are provided to help avoid pointer arithmetic mistakes.

PADD Perform unscaled pointer addition

```
static word_t* PADD(void* address, long distance) {
  return ((word_t*)((char*)(address) + (distance)));
}
```

PSUB Perform unscaled pointer subtraction

```
static word_t* PSUB(void* address, long distance) {
  return ((word_t*)((char*)(address) - (distance)));
}
```

Functions for pointer operations
The following functions can be used in place of pointer operations in your code to help detect errors early and avoid casting and pointer noise. These functions verify that all pointers used, stored, and loaded point within the heap and are word-aligned

LOAD(a) loads a word from memory at address a

```
static word_t LOAD(word_t* address) {
   assert(check_address(address) && “LOAD must load from a 16-byte
                                     aligned address within the heap");
   return *((word_t*)(address));
}
```

PLOAD(a) loads a pointer word from memory at address a

```
static word_t* PLOAD(word_t** address) {
   assert(check_address(address) && “PLOAD must load from a 16-byte
                                     aligned address within the heap");
 word_t* ptr = *((word_t**)(address));
 assert((!ptr || check_address(ptr)) && "PLOAD must return a 16-byte
                                     aligned address within the heap");
 return ptr;
}
```

STORE(a,w) stores word w into memory at address a

```
static void STORE(word_t* address, word_t word) {
     assert(check_address(address) && "STORE must store to a 16-byte
                                  aligned address within the heap");
     *((word_t*)address) = word;
}
```

PSTORE(a,w) stores pointer word w into memory at address a

```
static void PSTORE(word_t** address, void* ptr) {
   assert((!ptr || check_address(ptr))&& "PSTORE must store a 16-byte
                                 aligned address within the heap");
   assert(check_address(address) && "PSTORE must store to a 16-byte
                                 aligned address within the heap");
   *((word_t**)address) = ptr;
}
```

Traces

When learning about the starter code, you will simulate small **traces** (which are sequences of freeing and allocating blocks of various sizes). Results from the traces help you understand if your code is working correctly or not.

When testing and debugging, you may find it useful to write and test your own small traces.

A trace file contains 4 header lines:

Suggested heap size (any number, ignored by our tests).
Total number of blocks allocated.
Total number of malloc/free events.
Weight (any number, ignored by our tests).

Remaining lines after the header give a sequence of *memory management events* (either **free** or **allocate**), one per line. For example, the following example C code would generate the corresponding trace below it.

C code:

```
p0 = malloc(12);
p1 = malloc(16);
p2 = malloc(16);
free(p0);
free(p1);
p3 = malloc(24);
```

A corresponding trace:

```
128
4
6
1
a 0 12
a 1 16
a 2 16
f 0
f 1
a 3 24
```